SCALABLE ULTRA LIGHTWEIGHT COMPUTING THREADS

Inventors: Hong Zhang, Sheng Liang, Rajiv Mirani, and Abhishek Chauhan

Prepared By:
Edward A. Van Gieson, Reg. No. 44,386
Fenwick & West LLP
2 Palo Alto Square
Palo Alto, CA 94306
650.494.0600

Filed On: December 18, 2000

Express Mail No: EL444769597US

SCALABLE ULTRA LIGHTWEIGHT COMPUTING THREADS

Inventors: Hong Zhang, Sheng Liang, Rajiv Mirani, and Abhishek Chauhan

BACKGROUND OF THE INVENTION

5   ## 1. Field of the Invention

The present invention relates generally to scalable network computing services using computing threads. More particularly, the present invention is directed towards lightweight computing threads for scalable Internet services.

2. Description of Background Art

10   Many Internet web-sites and services use computer thread techniques to handle a large number of simultaneous user sessions. An individual thread is an independent unit of computation that forms a sequential execution stream. In many network applications a thread corresponds to the information required to serve one individual user or a particular service request. The request, in turn, may include one or more function calls

15   that are part of a code segment for performing the request. A thread scheduler is typically used to select a thread to be executed from a ready list of active threads. Threads may be scheduled based upon a prioritized list or by an un-prioritized round-robin technique.

A thread typically utilizes a hardware register, stack pointers, scheduling information, address space stack space, interrupt vectors, and other resources, depending upon the system. Execution of threads can be switched by saving values of CPU registers and other key data.

5   A thread uses stack, CPU register, and CPU time resources to execute a process. Threads are classified by their "weight." The weight of a thread corresponds to the amount of context that must be saved when a thread is removed from a processor and later restored. So-called lightweight threads include functionality for creating, deleting, scheduling, and synchronizing threads in a shared memory environment.

10   Highly parallel programs sometime use a thread-pool. A thread-pool is a collection of N threads that are used repeatedly to run small tasks. The threads execute in the same memory space, and can therefore work concurrently on shared data. A thread-pool thus serves as threading service in which a request from an interface is handed off to thread in the thread-pool.

15   Conventional thread techniques offer many advantages for multi-tasking a large number of user-sessions. However, network based services may be required to handle large numbers of concurrent user sessions. For example, a popular web-site may have to have the capacity to handle millions of concurrent user sessions. A firewall or proxy box may have to handle large numbers of concurrent connections. Streaming media and
20   media servers may have to serve a large number of audio and video streams corresponding to a large number of smaller processes.

A cost-effective network service is scalable, i.e., can be adapted to handle large numbers of concurrent user sessions, connections, or processes for a given amount of memory resources. Scalability for a large Internet server may be associated with the server having the capacity to handle millions of sessions, i.e., doubling the server

5   memory resources permits twice as many sessions. However, scalability can also refer to the ability of a device with limited memory resources to handle a large number of sessions relative to the memory resources of the device. For example, some compact network appliances have limited memory resources in order to reduce the size and cost of the network appliance. However, in the context of a scalable network appliance the

10   appliance should handle a large number of user sessions in proportion to the memory resources of the network appliance, i.e., a memory appliance with one-hundredth the memory resources of a large Internet server preferably handles a proportionately reduced number of simultaneous sessions.

Conventional thread techniques are not as scalable as desired. An active thread

15   may exist in several different states. The thread may be running a computational function using the CPU. Alternately, a thread may be waiting for another operation to occur, such as an input/output function. This is sometimes known as a "blocked" thread because the thread is active but waiting for non-CPU resources to complete its task. A blocked thread consumes memory, file descriptor, and other resources but is unavailable

20   to run code for other active sessions.

A system in which the threads are (on average) blocked for a significant fraction of their process time will require more threads to handle a given number of users. There

3   22271/01000/DOCS/1107228 4

are methods to switch from blocked threads to other threads. However, if context switches are performed, each context switch also consumes system resources. Each context switch has a system cost that is proportional to the size of the root thread context. The system cost of a context switch can be reduced, somewhat, by using a

5   cooperative threading model in which many of the entries in the register set do not need to be saved and restored. Storage of thread context also consumes stack memory resources. Threads are typically allotted a significant amount of virtual stack address space (e.g., 1 MB on the Win32 and 2MB on Solaris) so that each thread has the stack memory resource required to handle the functional calls in a session request.

10   Figure 1 is an illustrative diagram of a prior art thread based approach to map every user session onto a thread. Each user 102 has an outstanding user session 105, e.g., each user may be requesting information or viewing pages from a web-site. Each user session 105 is mapped onto a thread in a thread pool 110 using a thread pooling mechanism (not shown in Figure 1). Note that not all sessions are active at any given

15   time. A session is active only when it is carrying out a new task on the behalf of a user. For example, when a web email user clicks the "Check New Mail" button of an e-mail client, the user's session become active while checking for the availability of new e-mails. However, when a user is merely passively reading messages the e-mail session is not active. As a result, only active sessions need to be mapped to a thread.

20   Consequently, as shown in Figure 1, there can be more users 102 than threads 110.

Thread-based sessions work well for services in which there are comparatively small I/O delays to finish a thread session. As one example, thread-based sessions work

4   22271/01000/DOCS/1107228 4

well in web servers that serve static web pages from a disk. Each user session 105 for

receiving a static web page is a request that causes a thread 110 to read the requested

data from the disk and serve the data to the client. A disk read operation for a static web

page is completed comparatively quickly so that requests for static web pages can be

5   served in a short amount of time, i.e., the I/O delays are comparatively small.

However, thread-based sessions to not work well for services, such as web-

proxies, where the threads are blocked for significant periods of time because of input-

output (I/O) operations. A web proxy is used as an intermediary between a client and an

upstream server. In a typical situation, a request from the client causes the proxy to

10   issue a request to the upstream server. The proxy then waits for a response to come back

from the upstream server before issuing a response to the client. While the web proxy

waits for the response to come back from the upstream server it blocks an I/O operation

for an undetermined length of time. Extra threads are required to handle a given number

of sessions, increasing the stack memory resources required for the threads. Note also

15   that there are substantial system costs if a context switch is performed.

Figure 3 is a more detailed block diagram of a conventional thread system. A

user session 302 is mapped by a thread mapper 304 onto an individual thread 300. Each

thread 300 is allocated sufficient stack space 305 to handle any function the thread may

execute. Consequently, for a thread pool having a total of J threads, where J is an

20   integer, the total stack space allocated to the threads is the summation of the stack space

allocated to each of the J threads.

5   22271/01000/DOCS/1107228.4

An alternative conventional approach is the asynchronous event-based session shown in diagram of Figure 2. For the purposes of illustration, a single thread 210 is shown for executing each user session 105 of a group of simultaneous users 102. The asynchronous event-based session model has the advantage of reducing the time threads spend blocked but at the cost of a more complex programming model in which each session object is a state machine. In the asynchronous model the session objects must encode state transitions to permit a switch from a blocked session to another session. If a thread encounters a potentially blocking operation, such as an I/O operation, it records the current state of the session, suspends the current session, and goes on to pick up another session that needs to be processed. When the blocking operation finishes, the system generates an asynchronous event via asynchronous event generator 220 that marks the session runnable again. Each thread thus only blocks if there are no runnable sessions. This means that a small number of threads (e.g., one thread) can support a larger number of sessions.

The asynchronous model of Figure 2 works well for systems with stable and well defined protocols with well-defined state transitions, such as Internet firewalls and proxies. However, the programming model can comparatively complex because of the need to preserve state transition information. Moreover, if a new feature is added to the program, new states may have to be added to the state transition diagram. This can greatly increase the work required to upgrade a program, add new features, or modify existing features. The result is that the asynchronous model of FIG. 2 is both more

22271/01000/DOCS/1107228.4

difficult to program and more difficult to upgrade compared with the thread model of FIG. 1.

Conventional thread techniques thus do not have all of the desired features for handling a large number of user sessions. The thread-based model of FIG. 1 is

5 comparatively simple to program, but has problems with scalability, particularly if the threads are blocked because of I/O operations. Additional threads 110 can be added to the thread pool but there are significant system penalties associated with increasing the number of threads 110. This can limit the ability of a network device (e.g., a server or network appliance) to handle a large number of simultaneous sessions. The

10 asynchronous event-based model of FIG. 2 is highly scalable, but requires that the state of each session be recorded for blocked threads. This makes the asynchronous model more difficult to program and upgrade.

Therefore, there is a need for an new thread programming technique that is highly scalable and comparatively simple to program and upgrade.

15 SUMMARY OF THE INVENTION

The present invention permits a reduction in the stack memory allocated to a computer thread, which reduces the system penalty when large numbers of threads are used. In the present invention, each segment of program code to be run on a thread, such as the program code for executing an active network session, is classified based

20 upon the stack memory required to execute any function call within the segment The stack memory allocated to the thread is preferably sufficient to handle the requirements

of a first class of function calls. The computer thread utilizes an auxiliary stack resource to execute program code function calls requiring substantial stack memory resources and reclaims the auxiliary stack resource when it is no longer necessary to execute the code of the segment. In a wrapper embodiment, segments of the program code of a class having function calls requiring substantial stack resources are wrapped in a wrapper that calls an auxiliary stack resource for executing the function call and in which the wrapper returns the auxiliary stack resource when the function call is completed. The auxiliary stack resource may be a shared stack or a new stack from a pool of stacks.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a prior art thread pool for executing user sessions.

Figure 2 illustrates a prior art method to asynchronously execute user session in a threaded system.

Figure 3 illustrates a prior art thread system.

Figure 4A illustrates a thread system of the present invention in which auxiliary stack resources are called to handle peak stack memory requirements of complex function calls.

Figure 4B illustrates a program code having code segments.

Figure 4C illustrates a classification in which code is classified into two classes and four types.

22271/01000/DOCS/1107228.4

Figure 4D illustrates a program code in which program segment of a class requiring large stack resources are wrapped in a wrapper to call an auxiliary stack.

Figure 5 illustrates the flow of control in the system of Figure 4A for different code types.

5   Figure 6 illustrates a thread system in which a call is made to a new stack to execute a function call.

Figure 7 illustrates a thread system in which a call is made to a shared stack to execute a function call.

Figure 8A illustrates the flow of control for an exemplary program code.

10   Figure 8B illustrates how a single auxiliary stack can be multitasked to service several threads.

Figure 9 is a flow chart for programming user code.

The figures depict a preferred embodiment of the present invention for purposes of illustration only. One of skill in the art will readily recognize from the following

15   discussion that alternative embodiments of the structures and methods disclosed herein may be employed without departing from the principles of the claimed invention.

22271/01000/DOCS/1107228.4

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

FIG. 4A illustrates one aspect of the present invention. The program code of a user session 402 is mapped to a thread 400 by a thread mapper 404. The program code may include one or more function calls. Each function call may require a different amount of stack memory space. Consequently, during the execution of a single user session by one thread 400 the instantaneous stack resources required will depend upon the function call of the program code segment presently being run by thread 400.

Each thread 400 preferably includes a comparatively small amount of stack memory 405 scaled to handle the requirements of a first class of program code defined as being a code type that requires comparatively small amounts of stack space for its execution. An auxiliary stack 410 is called whenever the program code type being executed by thread 400 is of a second class, the second class being defined as being of a code type that requires larger amounts of stack space for its execution   This is illustrated in Figure 4A by code-dependent stack switcher 410, which calls an auxiliary stack 410 whenever the code is of a class requiring large stack memory resources.

As shown in FIG. 4B, a user code 470 may be divided into small code segments. As shown in FIG. 4C, user code 470 may be further divided into two classes of segments, a first class 450 having a comparatively small stack memory requirements and a second class 460 having a larger stack memory requirement. First class 450 may be further divided into two types, corresponding to a code type that doesn't block 490 and a code type that may block 492. Second class 460 may also be further divided into a code type that doesn't block 494 and a code type that may block 496.

22271/01000/DOCS/1107228.4

FIG. 5 shows a flow of control for the system of FIG. 4. When the code type is of the first class, i.e., types 490 and 492, the regular thread stack 405 is used by the thread to execute the program code. However, if the code type is of the second class, i.e., types 494 and 496, auxiliary stack 410 is called to execute the program code.

As shown in FIG. 4D, a preferred implementation of a stack switcher is a wrapper 472 for user code segments of the second class, with the wrapper adapted to call auxiliary stack resources sufficient to execute the wrapped function. FIG. 6 is a functional block diagram of one embodiment of the present invention in which a wrapper is used to call the auxiliary stack. User code 605, such as that for executing an active user session, has program code of the second class wrapped in a wrapper configured to call the auxiliary stack resource that is a new stack in a pool of stacks. If the function call of the code is of a code type in the class requiring large stack memory resources, a wrapper in the user code makes a call to a new stack from a stack pool 620 to execute the function call required to provide the desired service 630. FIG. 7 shows an alternate embodiment in which the wrapper makes a function call to a shared stack 720 to execute a user code of a type requiring a comparatively large stack memory space.

One aspect of the present invention is the dynamic allocation of stack memory in accord with the stack memory requirements of different types of function calls required to execute an active session. As shown in FIG. 5, whenever a function requiring substantial stack space (e.g., code of the second class 460, corresponding to code types 494 and 496) needs to be executed, an auxiliary stack 410 is called to execute the function and the auxiliary stack is reclaimed when the function is completed. Note that

22271/01000/DOCS/1107228.4

a reclaimed auxiliary stack 410 may be used by another thread, i.e., an auxiliary stack 410 may be multitasked to service more than one thread 400. Consequently, stack memory is utilized more efficiently than for a conventional thread 300 in which each thread 300 is allocated sufficient fixed stack memory 310 to handle any function call

5   that the thread may be required to execute in an active session.

As described above, a wrapper 472 is the preferred implementation of the stack switcher 420 used to call auxiliary stack memory space for the second class 460 of code types requiring substantial stack memory resources. Classification of code types into the first class 450 and the second class 460 can be performed in several different ways.

10   Many programming languages include hundreds or even thousands of different function calls. The stack memory required for a thread to execute a particular function may be: 1) estimated based upon the nature of the function call performed by the code; or 2) empirically tested to form a table of the stack memory requirements of a particular function.

15   Additionally, the stack usage of a function can be estimated using an iterative dataflow method. An exemplary iterative dataflow method to estimate stack usage utilizes two loop levels. The stack usage of a function in a program code segment, P, can be conservatively estimated (e.g., the estimate may be higher than the actual usage, but not lower) using the following iterative data flow process. Assume that a program

20   code segment, P, comprises functions {F1, F2, ..., Fn}. First, the stack usage of all functions is set to 0. Two loops are utilized, a level 1 loop and a level 2 loop. The level 1 loop repeatedly executes a level 2 loop until the inferred stack usage of {F1, F2, ...,

Fn} no longer changes, i.e., the process converges on an estimate of the inferred stack space. The level 2 loop performs for each function, F, in {F1, F2, ..., Fn} the following steps:

- If F is a leaf function, the amount of stack F uses is inferred to be the sum of amount of storage needed by its local variables.

- If F is not a leaf function, the amount of stack F needs is inferred to be the sum of storage needed by its local variables and the currently inferred stack space needed by other functions called by F. Calls to unknown functions (such as a function pointer bound at runtime) are assumed to require infinite stack space. When the above iterative process converges, the inferred stack space for each function is either a finite number which is a conservative estimate of the stack space needed by the function, or infinite (should the function call make calls to one or more unknown functions).

For the purpose of classifying code segment into classes based upon stack space requirements, a first stack memory level for thread stack memory 405 needs to be selected. This may be done in a variety of ways, including: 1) estimating a first stack memory level for which the majority of functional calls may be executed; 2) using statistical techniques to estimate a first stack memory level for which a preselected fraction of all function calls may be executed; or 3) using empirical or semi-empirical techniques to select a first stack memory level that results in the reduction of total stack memory requirements. A second stack memory level of the auxiliary stack 410 may be selected using a variety of techniques, such as: 1) estimating a second stack memory level for which any function call may be executed; and 2) using techniques to adjust the

22271/01000/DOCS/1107228 4

allocation of stack memory to auxiliary stack 410 to handle any function call of code of the second class.

Each segment of program code may be classified into a first class 450 having two types 490 and 492 and a second class having two types 494 and 496. The code types are summarized in TABLE 1, below. Type 1 code processes are types of code in which the thread 400 executes function calls that require insignificant stack memory resources and in which the thread does not block. Type 2 code processes are code types in which the thread executes function calls that require insignificant stack memory resources and in which the thread may be blocked. Type 3 code processes are code types in which the thread executes function calls that require significant stack memory resources and that do not become blocked. Finally, Type 4 code processes are processes in which the thread executes function calls that require significant stack memory resources and that may block.

STACK USAGE	DOESN'T BLOCK	MAY BLOCK
Code requires insignificant stack resources	Type 1 (e.g., simple library function calls)	Type 2 (e.g., direct system call)
Code requires significant stack resources	Type 3 (e.g., complex functions)	Type 4 (e.g., complex functions)

TABLE I. Table of code types.

22271/01000/DOCS/1107228.4

Examples of type 1 code processes include a sequence of code that does not

perform a function call or a sequence of code that includes calls to simple library

functions, such as the Unix program language functions *strcmp* or *memcpy*. In most

programming language there are only a comparatively small number of type 1 code

5   processes.

Type 2 code does not require significant stack space but may block. This type of

code may, for example, be a direct system call or a simple wrapper around a system call.

In the Unix program language, for example, there are 25 system calls that may block,

including: open, creat, close, read, readv, write,writev, poll, select, pipe, fcntl, ioctl, dup,

10   dup2, socket, accept, connect, recv, recvfrom, recvmsg, send, sendto, sendmsg,

gethostbyname, and gethostbyaddr. These exemplary Unix blocking functions are

interposed by threads, i.e., all calls to the function are redirected through a wrapper. The

wrapper registers the call with the thread package and suspends the current thread. The

thread package then carries out the call and resumes the execution of the thread that

15   initiated the call. An asynchronous event mechanism may be used to implement these

blocking calls. Thus, a thread issuing these blocking calls does not consume an

additional amount of stack space and simply blocks until the thread finishes the call.

Type 3 code corresponds to processes that consume a significant amount of stack

space during a portion of their execution but which do not block. Examples are code

20   processes that contain calls to complex functions, such as recursive functions, that may

require a large amount of stack to execute the function. One way to reduce the stack

memory required for the thread is by the use of auxiliary stack resources for executing

the complex function. The arguments from the current stack of the thread are copied

over to the auxiliary stack resources, the function is invoked, and the result is copied

back to the current stack of the thread. As described below in more detail, the auxiliary

stack resource may be an alternate stack or a stack pool, depending upon whether the

5   thread operates preemptively or cooperatively.

As one example, consider a pre-emptive thread having a function that requires

substantial peak stack resources. Referring to FIG. 6, a thread may obtain a new stack

from a pool of stacks, copy over the arguments, invoke the function, copy back the

result, and return the stack to the pool of stacks. As an illustrative example, suppose that

10   a thread executes a program code segment that calls the Unix function *malloc*. *Malloc* is

a function that may invoke complex internal memory operations, such as compacting the

memory or requesting more memory from the operating system. Consequently, using

conventional thread techniques, *malloc* typically requires that a large amount of stack

space be reserved, resulting in a heavy thread. However, in accord with one

15   embodiment of the present invention, the thread package to execute the *malloc* function

is written as a function called call_with_new_stack that is used wrap every cell of *malloc*.

For example, the code process:

Result=malloc(size);

Can be re-written as the wrapped function:

20   Result= *call_with_new_stack* (malloc, size).

The *call_with_new_stack* function wrapper may be implemented in a variety of different program languages as a function that creates a new stack of memory from a memory pool. The following Linux/Intel x86 assembly code segment is an example of how *call_with_new_stack* may be implemented:

5

```
1. call_with_new_stack:

2. pushl %ebp

3. movl %esp,%ebp

4. call alloc_stack

5. pushl %eax

6. movl %eax,%esp

7. addl $SHARED_STACK_SIZE,%esp

8. pushl 44(%ebp)

9. pushl 40(%ebp)

10. pushl 36(%ebp)

11. pushl 32(%ebp)

12. pushl 28(%ebp)

13. pushl 24(%ebp)
```

10

15

22271/01000/DOCS/1107228 4

14. pushl 20(%ebp)

15. pushl 16(%ebp)

16. pushl 12(%ebp)

17. movl 8(%ebp),%eax;

5   18. call *%eax

19. addl 36,%esp

20. movl %eax,%ebx

21. call free_stack

22. movl %ebx,%eax

10   23. leave

24. ret

The above code segment works as follows:

1) it saves the stack pointer %esp (lines 2-3);

2) it calls alloc_stack to allocate a stack segment (line 4);

15   3) it saves the returned stack address so that it can be freed later at line 21 (line

5);

22271/01000/DOCS/1107228 4

4) it resets the stack pointer to the new stack (lines 6-7);

5) it copies up to 9 arguments from the current stack to the new stack (lines 8-16); 5) it calls the target function (lines 17-18);

6) it pops off the 9 arguments so that the stack top points to the saved stack segment (line 19);

7) it saves the result of the target function call in %ebx. (line 20);

8) it frees the stack segment by calling `free_stack` (line 21);

9) it restore the target function call result (line 22); and 10) it returns the target function call result (lines 23-24).

The functions `alloc_stack` and `free_stack` implement a memory pool of stack segments:

void* alloc_stack(void);

void free_stack(void* stack);

Note that type 3 thread is non-blocking, which means that the function call will not be blocked. Consequently, the new stack will be used to execute the function and be reclaimed to the pool of stacks after the function is executed on the new stack. This means that the number of stacks in the pool of stacks may be much smaller than the number of type 3 threads, particularly if the thread executes the complex function (e.g., malloc) during only a portion of the process executed by the thread.

22271/01000/DOCS/1107228.4

Referring to FIG. 7, if the thread package is non-preemptive, a shared auxiliary stack segment 410 may be used instead of a stack pool. The non-blocking nature means that type 3 code will not result in a context switch. A shared alternate stack function can be wrapped in the wrapper code:

call_with_shared_stack.

As an example of a shared stack wrapper implementation is the following Linux/Intel x 86 assembly code segment:

1. Call_with_shared_stack:

2. Pushl %ebp

3. movl %esp, %ebp

4. movl $shared-stack, %esp

5. addl $shared_stack_size, %esp

6. pushl 44 (%ebp)

7. pushl 40 (%ebp)

8. pushl 36 (%ebp)

9. pushl 32 (%ebp)

10. pushl 28 (%ebp)

22271/01000/DOCS/1107228.4

11. pushl 24 (%ebp)

12. pushl 20 (%ebp)

13. pushl 16 (%ebp)

14. pushl 12 (%ebp)

15. movl 8 (%ebp), %eax;

16. call *%eax

17. leave

18. ret

The above code segment performs the following function:

1) saving the stack pointer, %esp (lines 2-3);

2) resetting the stack pointer to the shared large stack (lines 4-5);

3) copying up to six arguments from the current stack to the new stack (lines 6-14);

4) calling the function (lines 15-16); and

5) switching back the stack and returning the result (lines 17-18).

Type 4 code may consume a large amount of stack space and may also block. Examples of type 4 code are code segments that contain calls to complex functions, such

as recursive functions, that may in turn call blocking functions. As in type 3 threads, auxiliary stack resources may be used to execute the complex function using a wrapper. Because type 4 code blocks, a `call_with_new_stack` wrapper for calling a new stack from a stack pool is preferably used.

Type 4 code may also be advantageously re-written. With reference to FIG. 8A, the flow control of a user program code will spend a fraction of time, f, using the auxiliary stack that depends on the fraction of time that type 3 code executes and the fraction of time that type 4 code either executes or is blocked. With reference to FIG. 8B, an auxiliary stack 410 can be multitasked to service a large number of threads 400. Note that type 4 code may spend a significant period of time blocked. This means that type 4 code tends to hold onto the auxiliary stack 410, thereby reducing the number of threads that can be multitasked from one auxiliary stack 410. There is a significant benefit to re-writing type 4 functions as type 3 functions, i.e., converting comparatively simple type 4 blocking functions into type 3 non-blocking functions. Alternately, if the code cannot be rewritten to eliminate the blocking function, in some cases the code can be rewritten to have a reduced blockage time. Consequently, in a preferred embodiment, code having the longest potential blockage time is re-written to reduce the blockage time.

Referring again to FIGS. 8A and 8B, is also desirable to alter the program code to reduce the number of function calls that require a large size for auxiliary stack 410. The amount of stack memory space allocated for the auxiliary stack resources can be beneficially reduced by designing programs to minimize stack memory requirements

22   22271/01000/DOCS/1107228 4

during part or all of their execution. Programming techniques to reduce stack space usage include turning large stack allocated variables into heap allocated variables and turning recursive algorithms into iterative algorithms.

FIG. 9 is a flow chart of an algorithm for programming user code. First each

5   segment of the user code is classified 910 into the four code types 490, 492, 494, or 496. A determination 920 is preferably made whether the code is of a type that can be replaced with a more efficient alternative, such as replacing a type 4 blocking code with non-blocking code or replacing a recursive function with one using an iterative technique. If the code type can be beneficially rewritten, the code segment is modified

10   930. If the code segment is of type 3 or type 4 (second class) it is appropriately wrapped 940 to call an auxiliary stack. Process 900 can be implemented manually but is preferably implemented using a supervisory program having a database of code types and a set of rules for modifying the code segment 930.

A supervisory program to implement the method of FIG. 9 may characterize each

15   program code function call by running the function on a real or virtual system to determine the stack memory requirement of the function and whether the function is blocking or non-blocking. This information can be stored in a database. Appropriate wrappers for each function call can also be stored in the database for each function that can advantageously use the auxiliary stack memory. A supervisory program may be

20   programmed to automatically select an appropriate wrapper for each code segment of the second class 460. Additionally, the supervisory program may also store a database

22271/01000/DOCS/1107228.4

of function substitutions for type 4 blocking code that requires a large stack space or which blocks for a long period of time and make an appropriate replacement 930.

If a supervisory program is not used to automatically classify the code type of each segment of user code, the thread technique of the present invention requires that the code type (e.g., type 1, type 2, type 3, or type 4) be taken into account when programming the code in order to select an appropriate wrapper. The code type should also be taken into account in deciding whether to re-write the code to reduce potential blockage or to alter the function call to reduce stack memory requirements. The extra complexity increases the programming difficulty and chance for errors (e.g., using the wrong wrapper for a particular type of code or mistaking a blocking function for a non-blocking function). Several approaches may be used to reduce the chance of a programming error. One approach is to use a naming convention for functions that correspond to a particular code type. This allows the code type to be tracked by name. Run-time assertion check can also be used assist in catching code type errors.

Still another approach is to leverage built-in features of the programming language to reduce the chance of a human error occurring during the programming of the code. For example, if the programming language supports checked exceptions, such as JAVA, different types of code may be declared to throw different classes of checked exceptions. For example, functions that block may be declared such that they always take a particular type as an argument. These functions can then only be called in contexts where a variable of such type is available. Functions that block and use stack

may throw a `BlockingException`. For example, method `f` in class `C` may be declared as:

```
class C {

    f() throws BlockingException {...};

    ... }
```

5

Now any method that uses `C.f` will either: also throw `BlockingException` indicating the method also blocks and uses the stack, or have to catch and handle `BlockingException` explicitly. A blocking exception provides that the benefit of ensuring that an appropriate wrapper, such as `call_with_new_stack,` is used every

10 time `BlockingException` is handled.

While particular embodiments and applications of the present invention have been illustrated and described, it is to be understood that the invention is not limited to the precise construction and components disclosed herein and that various modifications, changes and variations which will be apparent to those skilled in the art may be made in

15 the arrangement, operation and details of the method and apparatus of the present invention disclosed herein without departing from the spirit and scope of the invention as defined in the appended claims.

22271/01000/DOCS/1107228.4